Our Ref.: 723-959

U.S. PATENT APPLICATION

Inventor(s): Farhad Fouladi
Robert Moore

Invention: METHOD AND APPARATUS FOR BUFFERING GRAPHICS DATA IN A GRAPHICS SYSTEM

NIXON & VANDERHYE P.C.
ATTORNEYS AT LAW
1100 NORTH GLEBE ROAD
8TH FLOOR
ARLINGTON, VIRGINIA 22201-4714
(703) 816-4000
Facsimile (703) 816-4100

SPECIFICATION

Method and Apparatus For Buffering Graphics Data In A Graphics System

This application claims the benefit of U.S. Provisional Application No. 60/226,912, filed August 23, 2000, the entire content of which is hereby incorporated by reference in this application.

Field of the Invention

The invention relates to computer graphics, and more particularly to interactive graphics systems such as home video game platforms. Still more particularly, this invention relates to efficient graphics command buffering between a graphics command producer and a graphics command consumer.

Background And Summary Of The Invention

Many of us have seen films containing remarkably realistic dinosaurs, aliens, animated toys and other fanciful creatures. Such animations are made possible by computer graphics. Using such techniques, a computer graphics artist can specify how each object should look and how it should change in appearance over time, and a computer then models the objects and displays them on a display such as your television or a computer screen. The computer takes care of performing the many tasks required to make sure that each part of the displayed image is colored and shaped just right based on the position and orientation of each object in a scene, the direction in which light seems to strike each object, the surface texture of each object, and other factors.

Because computer graphics generation is complex, computer-generated three-dimensional graphics just a few years ago were mostly limited to expensive specialized flight simulators, high-end graphics workstations and supercomputers.

The public saw some of the images generated by these computer systems in movies and expensive television advertisements, but most of us couldn't actually interact with the computers doing the graphics generation. All this has changed with the availability of relatively inexpensive 3D graphics platforms such as, for example, the Nintendo 64® and various 3D graphics cards now available for personal computers. It is now possible to interact with exciting 3D animations and simulations on relatively inexpensive computer graphics systems in your home or office.

A problem graphics system designers confronted in the past was how to efficiently buffer graphics commands between a graphics command producer and a graphics command consumer. Various solutions to this problem were offered. For example, it is well known to provide a buffer memory between a graphics command producer and a graphics command consumer. Often, this buffer memory is connected as part of the graphics command consumer (for example, on board a graphics chip). The graphics command producer writes graphics commands into the buffer memory, and the graphics command consumer reads those graphics commands from the buffer memory. It is typical for such a buffer memory to be structured as a first-in-first-out (FIFO) buffer so that the graphics command consumer reads the graphics command in the same sequence that they were written into the buffer by the graphics command producer.

Placing such a buffer between the producer and the consumer relaxes the degree to which the producer and consumer must be synchronized. The producer can write commands into the buffer at an instantaneous rate that is independent of the instantaneous rate at which the consumer reads commands from the buffer. Even if the consumer suffers a momentary delay in reading from the buffer (e.g., as may occur when the producer asks the consumer to draw large or complex

primitives), the producer will not stall unless/until it fills the buffer and has no more memory space to write new commands. Similarly, momentary delays of the producer in writing new graphics commands into the buffer will not cause the consumer to stall unless the consumer consumes all of the graphics commands in the buffer before the producer has an opportunity to write additional graphics commands.

A potential problem encountered in the past relates to the size of the buffer. Because of limitations on chip size and complexity, it is often not possible to put a very large command buffer memory on the graphics chip. A small sized FIFO buffer in the graphics hardware may not adequately load balance between the producer and the consumer, causing the producer to stall when the consumer renders big primitives. Thus, while significant work has been done in the past, further improvements are possible.

The present invention solves this problem by providing techniques and arrangements that more efficiently buffer graphics commands between a graphics command producer and a graphics command consumer. In accordance with one aspect of the invention, a part of main memory shared between the producer and consumer is allocated to a variable number of variable sized graphics command buffers. The producer can specify the number of buffers and the size of each. Writes to the graphics consumer can be routed to any of the buffers in main memory. A buffer can be attached simultaneously to the consumer and the producer, or different buffers can be attached to the consumer and the producer. In the multi-buffering approach where different buffers are attached to the consumer and the producer, the producer can write to one buffer while the consumer reads from another buffer.

To further decouple the consumer from the producer, the producer and consumer independently maintain their own read and write pointers in accordance with another aspect of the invention. Even though the consumer may not write to the buffer, it nevertheless maintains a write pointer which it uses to keep track of data valid position within the buffer. Similarly, even though the producer may not read from the buffer it is attached to, it maintains a read pointer which it uses to keep track of data valid position within the buffer. The effect of this pointer arrangement is to further decouple the producer from the consumer --reducing the synchronization requirements between the two.

In accordance with another aspect provided by this invention, the producer can write a "call display list" command to a FIFO buffer that directs the consumer to read a string of graphics commands (e.g., a display list) stored elsewhere in memory, and to subsequently return to reading the rest of the buffer. This ability to call an out-of-line graphics command string from a FIFO buffer provides additional flexibility and further decreases synchronization requirements.

In accordance with another aspect of the invention, the graphics command producer can write a graphics command stream to a FIFO buffer that includes a command which automatically redirects succeeding commands to a display list buffer. One way to visualize this is to picture the graphics command producer as a redirectable fire hose that continually produces a stream of graphics commands. The fire hose normal streams the graphics command into a FIFO buffer. However, the producer can include, within the stream, a "Begin Display List" command that causes graphics commands following the command to be written to a display list instead. An "End Display List" command inserted further on in the stream can terminate the display list and redirect the graphics command stream back to the

same (or different) FIFO buffer. This feature has the advantage of allowing the graphics command producer to efficiently create reusable display lists with very low overhead.

In accordance with another aspect provided by this invention, the graphics command producer can insert a break point into any of multiple FIFO buffers. The break point can cause the consumer to interrupt. Such break points can help to synchronize the producer and the consumer when close synchronization is required.

In accordance with yet another aspect provided by this invention, the graphics system includes a producer that outputs graphics commands, a consumer that consumes the graphics commands outputted by the producer, and a storage device coupled between the producer and the consumer. The storage device stores plural variable sized buffers disposed at variable locations within the storage device. Each of the variable sized buffers receives and temporarily stores graphics commands outputted by the producer for delivery to the consumer.

In accordance with a further aspect provided by the invention, the consumer is incapable of writing to at least an active one of the plural buffers, but nevertheless maintains – independently of the producer – a write pointer for at least the active one of the plural buffers. The producer provides a producer read pointer and a producer write pointer associated with a first of the plural buffers, and the consumer independently maintains a consumer read pointer and a consumer write pointer associated with that same buffer. The consumer may increment the consumer read pointer as the consumer reads from an active buffer and suspends reading from the active buffer when the incremented consumer read pointer has a predetermined relationship with a consumer write pointer. The consumer may selectively

increment the consumer write pointer in response to the producer writing to the active buffer.

In accordance with another aspect of the invention, a buffer includes a read command that controls the consumer to consume a set of graphics commands the producer stores elsewhere within the storage device, and to resume consuming graphics commands from the buffer after consuming the graphics commands stored elsewhere. The read command may specify a starting address and a length of a display list. The read command controls the consumer to read the display list of the specified length beginning at the specified starting address.

In accordance with another aspect of the invention, any of the plural buffers may provide either circular or linear first-in-first-out access.

In accordance with another aspect of the invention, any of the plural buffers can be selectively attached to both the producer and the consumer simultaneously – or one of the buffers can be attached to the producer while another buffer is attached to the consumer.

In accordance with still another aspect provided by the invention, the producer allocates the size of each of the plural buffers. Such allocation is provided so that each buffer is capable of storing at least a frame of graphics commands.

In accordance with another aspect of the invention, the producer may write a break point into any of the plural buffers. The consumer may suspend consumption of graphics commands upon encountering the break point.

In accordance with yet another aspect of the invention, each buffer may provide an overflow status indicator indicating when the producer overwrites a location in the buffer.

In accordance with yet another aspect of the invention, a status register or other indicator may indicate the status of at least one of the plural buffers. The status register may indicate, for example:

- producer writer pointer position,
- producer read pointer position,
- consumer write pointer position, and
- consumer read pointer position.

In accordance with yet another aspect provided by this invention, a graphics system includes:

- a storage buffer that receives and temporarily stores graphics commands,
- a producer that writes graphics commands into the buffer, the producer maintaining a producer write pointer and a producer read pointer associated with the buffer, and
- a consumer that consumes graphics commands stored within the buffer, the consumer maintaining a consumer write pointer that is independent of the producer write pointer and a consumer read pointer that is independent of the producer read pointer.

In accordance with yet another aspect of this invention, a graphics system includes a graphics command producer that writes graphics commands into a buffer based on a producer write pointer, and a graphics commands consumer that reads graphics commands from the buffer based on a consumer read pointer. In accordance with this aspect of the invention, the consumer write pointer is independently maintained by the consumer and indicates the extent of valid data the producer has written into the buffer. The consumer ceases to consume graphics

commands from the buffer upon the consumer read pointer having a predetermined relationship to the consumer write pointer.

In accordance with yet another aspect provided by this invention, an interactive graphics system includes a processor module executing an application, a graphics processor module, and at least one memory coupled to the processor module and to the graphics processor module. The method of controlling the flow of graphics commands between the processor module and the graphics processor module comprises:

- dynamically establishing, under control of the application, a variable number of FIFO buffers in the memory, the application specifying the size of each of the FIFO buffers,
- the application controlling the processor module to write graphics commands into at least a first of the plurality of FIFO buffers, and
- the application sending graphics commands to the graphics processor module that control the graphics processor module to read graphics commands from the first FIFO buffer.

The processor module may provide a processor module read pointer and processor module write pointer associated with the first of plurality of buffers. The graphics processor module may independently maintain a graphics processor module read pointer and a graphics processor module write pointer associated with the first buffer. The graphics processor module may increment the graphics processor read pointer each time the graphics processor module reads from the first buffer, and may suspend reading from the first buffer when the graphics processor module read pointer has a predetermined relationship with the graphics processor module write pointer. Graphics processor module may selectively auto increment

the graphics processor write pointer in response to the processor writing to the first buffer.

In accordance with yet another aspect of the invention, a method of controlling the flow of graphics data comprises:

- writing graphics data into plural variable sized FIFO buffers each having plural storage locations,
- setting a break point associated with at least one of the plural storage locations,
- reading graphics data from the plural buffers in a predetermined order,
- temporarily suspending the reading step upon encountering the at least one location associated with the break point, and generating an interrupt, and
- resuming the reading step in response to receipt of a clear interrupt command.

In accordance with yet another aspect provided by this invention, a graphics system includes:

- a storage device that receives and temporarily stores graphics commands,
- a producer that writes commands into a buffer within the storage device, the commands including a first set of graphics commands and a read command referring to a second set of graphics commands stored elsewhere in the storage device, and
- a consumer that consumes the first set of graphics commands stored within the buffer and, in response to encountering the read

command, consumes the second set of graphics commands and subsequently consumes additional commands from the buffer.

Brief Description Of The Drawings

These and other features and advantages provided by the invention will be better and more completely understood by referring to the following detailed description of presently preferred embodiments in conjunction with the drawings, of which:

Figure 1 is an overall view of an example interactive computer graphics system;

Figure 2 is a block diagram of the Figure 1 example computer graphics system;

Figure 3 is a block diagram of the example graphics and audio processor shown in Figure 2;

Figure 4 is a block diagram of the example 3D graphics processor shown in Figure 3;

Figure 5 is an example logical flow diagram of the Figure 4 graphics and audio processor;

Figure 6 shows example multi-buffering;

Figure 7 shows example independent consumer and producer read and write pointers;

Figures 8A and 8B show, respectively, example empty and full buffer conditions;

Figure 9 shows an example call of a display list from an FIFO buffer;

Figures 10A-10C show example display list creation; and

Figure 11 shows an example FIFO manager implementation; and

Figures 12A and 12B show example alternative compatible implementations.

Detailed Description Of Example Embodiments Of The Invention

Figure 1 shows an example interactive 3D computer graphics system 50. System 50 can be used to play interactive 3D video games with interesting stereo sound. It can also be used for a variety of other applications.

In this example, system 50 is capable of processing, interactively in real time, a digital representation or model of a three-dimensional world. System 50 can display some or all of the world from any arbitrary viewpoint. For example, system 50 can interactively change the viewpoint in response to real time inputs from handheld controllers 52a, 52b or other input devices. This allows the game player to see the world through the eyes of someone within or outside of the world. System 50 can be used for applications that do not require real time 3D interactive display (e.g., 2D display generation and/or non-interactive display), but the capability of displaying quality 3D images very quickly can be used to create very realistic and exciting game play or other graphical interactions.

To play a video game or other application using system 50, the user first connects a main unit 54 to his or her color television set 56 or other display device by connecting a cable 58 between the two. Main unit 54 produces both video signals and audio signals for controlling color television set 56. The video signals are what controls the images displayed on the television screen 59, and the audio signals are played back as sound through television stereo loudspeakers 61L, 61R.

The user also needs to connect main unit 54 to a power source. This power source may be a conventional AC adapter (not shown) that plugs into a standard home electrical wall socket and converts the house current into a lower DC voltage signal suitable for powering the main unit 54. Batteries could be used in other implementations.

The user may use hand controllers 52a, 52b to control main unit 54. Controls 60 can be used, for example, to specify the direction (up or down, left or right, closer or further away) that a character displayed on television 56 should move within a 3D world. Controls 60 also provide input for other applications (e.g., menu selection, pointer/cursor control, etc.). Controllers 52 can take a variety of forms. In this example, controllers 52 shown each include controls 60 such as joysticks, push buttons and/or directional switches. Controllers 52 may be connected to main unit 54 by cables or wirelessly via electromagnetic (e.g., radio or infrared) waves.

To play an application such as a game, the user selects an appropriate storage medium 62 storing the video game or other application he or she wants to play, and inserts that storage medium into a slot 64 in main unit 54. Storage medium 62 may, for example, be a specially encoded and/or encrypted optical and/or magnetic disk. The user may operate a power switch 66 to turn on main unit 54 and cause the main unit to begin running the video game or other application based on the software stored in the storage medium 62. The user may operate controllers 52 to provide inputs to main unit 54. For example, operating a control 60 may cause the game or other application to start. Moving other controls 60 can cause animated characters to move in different directions or change the user's point of view in a 3D world. Depending upon the particular software stored within the storage medium 62, the

various controls 60 on the controller 52 can perform different functions at different times.

Example Electronics of Overall System

Figure 2 shows a block diagram of example components of system 50. The primary components include:

- a main processor (CPU) 110,
- a main memory 112, and
- a graphics and audio processor 114.

In this example, main processor 110 (e.g., an enhanced IBM Power PC 750) receives inputs from handheld controllers 108 (and/or other input devices) via graphics and audio processor 114. Main processor 110 interactively responds to user inputs, and executes a video game or other program supplied, for example, by external storage media 62 via a mass storage access device 106 such as an optical disk drive. As one example, in the context of video game play, main processor 110 can perform collision detection and animation processing in addition to a variety of interactive and control functions.

In this example, main processor 110 generates 3D graphics and audio commands and sends them to graphics and audio processor 114. The graphics and audio processor 114 processes these commands to generate interesting visual images on display 59 and interesting stereo sound on stereo loudspeakers 61R, 61L or other suitable sound-generating devices.

Example system 50 includes a video encoder 120 that receives image signals from graphics and audio processor 114 and converts the image signals into analog and/or digital video signals suitable for display on a standard display device such as

a computer monitor or home color television set 56. System 50 also includes an audio codec (compressor/decompressor) 122 that compresses and decompresses digitized audio signals and may also convert between digital and analog audio signaling formats as needed. Audio codec 122 can receive audio inputs via a buffer 124 and provide them to graphics and audio processor 114 for processing (e.g., mixing with other audio signals the processor generates and/or receives via a streaming audio output of mass storage access device 106). Graphics and audio processor 114 in this example can store audio related information in an audio memory 126 that is available for audio tasks. Graphics and audio processor 114 provides the resulting audio output signals to audio codec 122 for decompression and conversion to analog signals (e.g., via buffer amplifiers 128L, 128R) so they can be reproduced by loudspeakers 61L, 61R.

Graphics and audio processor 114 has the ability to communicate with various additional devices that may be present within system 50. For example, a parallel digital bus 130 may be used to communicate with mass storage access device 106 and/or other components. A serial peripheral bus 132 may communicate with a variety of peripheral or other devices including, for example:

- a programmable read-only memory and/or real time clock 134,
- a modem 136 or other networking interface (which may in turn connect system 50 to a telecommunications network 138 such as the Internet or other digital network from/to which program instructions and/or data can be downloaded or uploaded), and
- flash memory 140.

A further external serial bus 142 may be used to communicate with additional expansion memory 144 (e.g., a memory card) or other devices. Connectors may be used to connect various devices to busses 130, 132, 142.

Example Graphics And Audio Processor

Figure 3 is a block diagram of an example graphics and audio processor 114. Graphics and audio processor 114 in one example may be a single-chip ASIC (application specific integrated circuit). In this example, graphics and audio processor 114 includes:

- a processor interface 150,
- a memory interface/controller 152,
- a 3D graphics processor 154,
- an audio digital signal processor (DSP) 156,
- an audio memory interface 158,
- an audio interface and mixer 160,
- a peripheral controller 162, and
- a display controller 164.

3D graphics processor 154 performs graphics processing tasks. Audio digital signal processor 156 performs audio processing tasks. Display controller 164 accesses image information from main memory 112 and provides it to video encoder 120 for display on display device 56. Audio interface and mixer 160 interfaces with audio codec 122, and can also mix audio from different sources (e.g., streaming audio from mass storage access device 106, the output of audio DSP 156, and external audio input received via audio codec 122). Processor interface 150

provides a data and control interface between main processor 110 and graphics and audio processor 114.

Memory interface 152 provides a data and control interface between graphics and audio processor 114 and memory 112. In this example, main processor 110 accesses main memory 112 via processor interface 150 and memory interface 152 that are part of graphics and audio processor 114. Peripheral controller 162 provides a data and control interface between graphics and audio processor 114 and the various peripherals mentioned above. Audio memory interface 158 provides an interface with audio memory 126.

Example Graphics Pipeline

Figure 4 shows a more detailed view of an example 3D graphics processor 154. 3D graphics processor 154 includes, among other things, a command processor 200 and a 3D graphics pipeline 180. Main processor 110 communicates streams of data (e.g., graphics command streams and display lists) to command processor 200. Main processor 110 has a two-level cache 115 to minimize memory latency, and also has a write-gathering buffer 111 for uncached data streams targeted for the graphics and audio processor 114. The write-gathering buffer 111 collects partial cache lines into full cache lines and sends the data out to the graphics and audio processor 114 one cache line at a time for maximum bus usage.

Command processor 200 receives display commands from main processor 110 and parses them -- obtaining any additional data necessary to process them from shared memory 112. The command processor 200 provides a stream of vertex commands to graphics pipeline 180 for 2D and/or 3D processing and rendering. Graphics pipeline 180 generates images based on these commands. The resulting

image information may be transferred to main memory 112 for access by display controller/video interface unit 164 -- which displays the frame buffer output of pipeline 180 on display 56.

Figure 5 is a logical flow diagram of graphics processor 154. Main processor 110 may store graphics command streams 210, display lists 212 and vertex arrays 214 in main memory 112, and pass pointers to command processor 200 via bus interface 150. The main processor 110 stores graphics commands in one or more graphics first-in-first-out (FIFO) buffers 210 it allocates in main memory 110. The command processor 200 fetches:

- command streams from main memory 112 via an on-chip FIFO memory buffer 216 that receives and buffers the graphics commands for synchronization/flow control and load balancing,
- display lists 212 from main memory 112 via an on-chip call FIFO memory buffer 218, and
- vertex attributes from the command stream and/or from vertex arrays 214 in main memory 112 via a vertex cache 220.

Command processor 200 performs command processing operations 200a that convert attribute types to floating point format, and pass the resulting complete vertex polygon data to graphics pipeline 180 for rendering/rasterization. A programmable memory arbitration circuitry 130 (see Figure 4) arbitrates access to shared main memory 112 between graphics pipeline 180, command processor 200 and display controller/video interface unit 164.

Figure 4 shows that graphics pipeline 180 may include:

- a transform unit 300,

- a setup/rasterizer 400,
- a texture unit 500,
- a texture environment unit 600, and
- a pixel engine 700.

Transform unit 300 performs a variety of 2D and 3D transform and other operations 300a (see Figure 5). Transform unit 300 may include one or more matrix memories 300b for storing matrices used in transformation processing 300a. Transform unit 300 transforms incoming geometry per vertex from object space to screen space; and transforms incoming texture coordinates and computes projective texture coordinates (300c). Transform unit 300 may also perform polygon clipping/culling 300d. Lighting processing 300e also performed by transform unit 300b provides per vertex lighting computations for up to eight independent lights in one example embodiment. Transform unit 300 can also perform texture coordinate generation (300c) for embossed type bump mapping effects, as well as polygon clipping/culling operations (300d).

Setup/rasterizer 400 includes a setup unit which receives vertex data from transform unit 300 and sends triangle setup information to one or more rasterizer units (400b) performing edge rasterization, texture coordinate rasterization and color rasterization.

Texture unit 500 (which may include an on-chip texture memory (TMEM) 502) performs various tasks related to texturing including for example:

- retrieving textures 504 from main memory 112,
- texture processing (500a) including, for example, multi-texture handling, post-cache texture decompression, texture filtering, embossing, shadows

and lighting through the use of projective textures, and BLIT with alpha transparency and depth,

- bump map processing for computing texture coordinate displacements for bump mapping, pseudo texture and texture tiling effects (500b), and
- indirect texture processing (500c).

Texture unit 500 outputs filtered texture values to the texture environment unit 600 for texture environment processing (600a). Texture environment unit 600 blends polygon and texture color/alpha/depth, and can also perform texture fog processing (600b) to achieve inverse range based fog effects. Texture environment unit 600 can provide multiple stages to perform a variety of other interesting environment-related functions based for example on color/alpha modulation, embossing, detail texturing, texture swapping, clamping, and depth blending..

Pixel engine 700 performs depth (z) compare (700a) and pixel blending (700b). In this example, pixel engine 700 stores data into an embedded (on-chip) frame buffer memory 702. Graphics pipeline 180 may include one or more embedded DRAM memories 702 to store frame buffer and/or texture information locally. Z compares 700a' can also be performed at an earlier stage in the graphics pipeline 180 depending on the rendering mode currently in effect (e.g., z compares can be performed earlier if alpha blending is not required). The pixel engine 700 includes a copy operation 700c that periodically writes on-chip frame buffer 702 to main memory 112 for access by display/video interface unit 164. This copy operation 700c can also be used to copy embedded frame buffer 702 contents to textures in the main memory 112 for dynamic texture synthesis effects. Anti-aliasing and other filtering can be performed during the copy-out operation. The frame buffer output of graphics pipeline 180 (which is ultimately stored in main

memory 112) is read each frame by display/video interface unit 164. Display controller/video interface 164 provides digital RGB pixel values for display on display 102.

FIFO Buffers Allocated In Shared Memory

In this example, the command FIFO buffer 216 (which may be a small dual ported RAM streaming buffer) on board the graphics and audio processor 114 is too small, by itself, to do a good job of load balancing between the processor 110 and the graphics pipeline 180. This may result in the processor 110 becoming stalled when the graphics and audio processor 114 is rendering big primitives. To remedy this problem, we use part of the main memory 112 shared between processor 110 and graphics and audio processor 114 as a command FIFO buffer 210. The use of buffers 210 allows the main processor 110 and the graphics processor 114 to operate in parallel at close to their peak rates.

There are (at least) two methods of using buffers 210 to achieve parallelism: immediate mode and multi-buffer mode. When a single buffer 210 is attached to both the main processor 110 and the graphics processor 114, the system 50 is operating in the immediate mode. As the main processor 110 writes graphics commands to the buffer 210, the graphics processor 114 processes them in order. Hardware support provides flow control logic to prevent writes from overrunning reads and to wrap the read and write pointers of the buffer 210 back to the first address to provide circular buffer operation.

In the preferred embodiment, it is also possible to connect one buffer 210 to the main processor 110 while the graphics and audio processor 114 is reading from a different buffer 210(1) in a multi-buffered mode. In this case, the buffers 210(1),

210(2) are managed more like buffers than traditional FIFOs since there are no simultaneous reads and writes to any particular buffer 210. Multi-buffer mode may be used, for example, if dynamic memory management of the buffers is desirable.

Figure 6 shows how a portion of shared memory 112 can be allocated to provide multiple FIFO command buffers 210(1), 210(2), ..., 210(n) to buffer graphics (and audio) commands between the producer 110 and the consumer 114. In the example shown in Figure 6, each of buffers 210 receives graphics (and/or audio) commands from main processor 110, and provides those commands to graphics and audio processor 114. Main processor 110 allocates portions of main memory 112 for use as these buffers 210. A buffer data structure describing a region of main memory can be allocated by an application running on main processor 110.

Main processor 110 writes graphics commands into the buffers using a write pointer 802. Graphics and audio processor 114 reads commands from buffers 210 using a read pointer 804. Write pointer 802 and read pointer 804 can point to the same or different buffers. In this way, the same buffer 210 may be "attached" to both the main processor 110 and the graphics and audio processor 114 simultaneously – or different buffers may be attached to the producer and consumer at different times.

In the multi-buffering example shown in Figure 6, the main processor 110 and the graphics and audio processor 114 don't necessarily agree on where "the" FIFO buffer 210 is located. In the example shown, the main processor 110 is using buffer 210(2) as its current buffer for writing graphics commands to, whereas the graphics and audio processor 114 uses a different buffer 210(1) as its current buffer for obtaining graphics commands. Buffers 210 can be dynamically attached to main processor 110, graphics and audio processor 114, or both. When a buffer is

attached to the main processor 110, the main processor will write graphics commands into the buffer 210. In the example embodiment, there is always one and only one buffer 210 attached to main processor 110 at any one time. When a buffer 210 is attached to the graphics processor 114, the graphics processor will read and process graphics commands from the attached buffer 210. Only one buffer 210 can be attached to the graphics processor 114 at any one time in this example.

Independent Consumer and Producer Read and Write Pointers

Even though main processor 110 acting as graphics command producer does not need to read from the buffer 210(2) to which it is attached, it nevertheless maintains a producer read pointer 806 in this Figure 6 example. Similarly, even though the graphics and audio processor 114 acts as a consumer of graphics commands and therefore does not need to write to the buffer 210(1) to which it is attached, it nevertheless maintains a consumer write pointer 808 in the Figure 6 example. These additional pointers 806, 808 allow the producer and consumer to independently maintain the respective buffer 210 to which it is attached.

The additional pointer 806 maintained by main processor 110 and the additional pointer 808 maintained by graphics and audio processor 114 are used to provide overlap detection. These extra pointers indicate where valid data exists within the buffer 210. For example, the main processor 110 may treat the buffer 210(2) to which it is attached as a circular buffer, and "wrap" its write pointer around to the "beginning" of the buffer 810 once it reaches the "end" of the buffer 812. However, once the producer write pointer 802 encounters the producer read pointer 806, it will cease writing to attached buffer 210(2) to avoid overwriting valid, previously written data that the graphics and audio processor 114 has not yet read. Similarly, the graphics and audio processor consumer 114 may continue to

increment its read pointer 804 as it progressively reads graphics instructions from its attached buffer 210(1), but will cease this incrementing procedure when the read pointer 804 encounters the write pointer 808 – since the consumer is using the write pointer as indicating the last valid data within the buffer 210(1).

Pointers 802, 804, 806, and 808 can point to any location within buffers 210. Valid data may thus exist anywhere within these buffers – not necessarily at the beginning or at the end of the buffer. In fact, if buffers 210 are operated in a circular mode, there is no concept of "beginning" or "end" since the end of the buffer wraps around to the beginning and the buffer is therefore a logically continuous loop.

Figure 7 provides a simplified explanation of the independent consumer and producer read and write pointers. In the Figure 7 example, consumer 114 uses an auto-incrementing read pointer 804 to read graphics commands from the buffer 210(1) to which it is attached. Consumer 114 also maintains a consumer write pointer 808 that points to the last valid graphics command within buffer 210(1). In this example, consumer 114 will continue to read graphics commands from buffer 210(1), and increment its read pointer 804 after each graphics command read, until the read pointer points to the same location that the write pointer points to (see Figure 8A). When the consumer 114 has incremented its read pointer 804 so that it points to the location adjacent the one that the write pointer 808 points to, the consumer "knows" that it has read all of the valid graphics commands from buffer 210(1) and has thus emptied the buffer. This condition indicates that the consumer 114 either needs to wait for more graphics commands from producer 110 (if the buffer 210(1) is also attached to the producer simultaneously), or it needs direction

as to a different buffer 210 it should begin reading from (if multi-buffering is in effect).

Similarly, the producer 110 may continue to write graphics commands into its attached buffer 210(2) and similarly continues to auto-increment its producer write pointer 802 until the write pointer points to the location in the buffer that is just before the location the producer read pointer 806 points to (see Figure 8B). In this example, coincidence (actually, close proximity) between the write pointer 802 and read pointer 806 indicates that the buffer 210(2) is full. If multi-buffering is in effect, producer 110 may at this point cease writing to buffer 210(2) and "save" (close) it, instruct the consumer 114 to read (now or later) the contents of that "closed" buffer, and begin writing additional graphics commands to yet another buffer 210 it can allocate within main memory 112. If the producer 110 and consumer 114 are attached to the same buffer 210, then the producer may need to wait until the consumer reads some commands before writing any more commands to the buffer. As explained below, to avoid frequent context switching, the preferred embodiment can provides a programmable hysteresis effect that requires the buffer to be emptied by a certain amount before the producer 110 is allowed to resume writing to the buffer, and requires the buffer to be filled by a certain amount before the consumer is allowed to resume reading from the buffer.

In the preferred embodiment, the main processor 110 writes graphics commands to the buffer 210 to which it is attached in 32-byte transfers. Main processor 110 provides a write-gathering buffer/function 111 (see Figure 4) that automatically packs graphics commands into 32-byte words. Graphics processor 114 reads graphics commands from the buffer 210 to which it is attached in 32-byte transfers.

Call Display List From FIFO Buffer

Figure 9 shows an example technique provided by the preferred example embodiment whereby an entry in a FIFO buffer 210 can call a display list – almost as if it were a function call. In this example, a command 890 is inserted into the graphics command FIFO 210 that calls a display list 212 stored elsewhere in memory. Upon encountering this command 890, the graphics processor 114 temporarily ceases reading graphics commands from FIFO buffer 210 and instead begins reading commands from a display list 212 stored elsewhere in main memory 112. Upon reaching the end of the display list 212, the graphics processor 114 returns to read the next sequential command from the graphics FIFO 210. This technique is quire useful in allowing multiple frames to call the same display list 212 (e.g., to render geometry which remains static from frame to frame) without requiring the main processor 112 to rewrite the display list for each frame.

Figures 10A through 10C show how main processor 110 can automatically create a display list 212 by writing to a graphics command FIFO 210. As shown in Figure 10A, main processor 110 begins by writing a graphics command stream to a graphics command FIFO 210 it allocates in main memory 112. At any point in this writing process, the main processor 110 can insert a "Begin Display List" command 890 into the FIFO buffer 210 that causes further writes from the main processor to be directed to a display list 212. Figure 10C shows that once main processor 110 is finished writing display list 212, it may issue an "End Display List" command that has the effect of automatically terminating the display list and redirecting the main processor command stream output back to FIFO buffer 210. One can visualize main processor 110 providing a redirectable "fire hose" command stream output that can gush graphics commands into FIFO buffer 210, display list 212, and back to the

same or different FIFO buffer 212. The display lists 212 created in this manner can remain in memory 112 and reused for parts of images that remain static over several frames or frame portions.

Example Implementation Details

A processor to graphics interface unit portion 202 of the graphics and audio processor 114 command processor 200 contains the control logic for managing the FIFO buffers 210 in main memory 112. Figure 11 shows an example implementation. In the example shown, all CPU 110 writes to the graphics and audio processor 114 will be routed to the main memory 112. There are two registers that define the portion of the main memory 112 that has been allocated to the graphics FIFO 210 attached to the graphics and audio processor 114:

the FIFO BASE register 822, and

the FIFO TOP register 824.

The FIFO_BASE register 822 defines the base address of the FIFO 210. The FIFO_TOP register 824 defines the last address in the FIFO.

Command processor 200 keeps track of the read and write pointers for FIFO 210 in hardware. Since all data written into the FIFO are cache line sized, there is no need to keep track of valid bytes. The write pointer 808 is incremented by 32 bytes every a cache line is written to an address that is between FIFO_BASE and FIFO_TOP (5LSBs are 0). Reading of the FIFO 210 is also performed one cache line at a time. The read pointer is incremented by 32 after a cache line has been read.

Initially, read pointer 804 and write pointer 808 are initialized to point to the same location, which means the FIFO is empty (see Figure 8A). The FIFO full

condition is (read pointer – 1) = (write pointer) (see Figure 8B). Write pointer 808 wraps around to the FIFO_BASE 204(2) address after it reaches FIFO_TOP. The read pointer 804 also wraps around when it reaches FIFO_TOP 824. The read pointer 804 is controlled by the hardware to make sure it doesn't get ahead of the write pointer 808, even in the wrap around cases. The application running on processor 110 makes sure that the write pointer 808 doesn't surpass the read pointer 804 after wrapping around.

Data from two (or more) different frames can be resident in the same FIFO 210. A break point mechanism can be used to prevent the command processor 200 from executing the second frame before the first frame can be copied out of the embedded DRAM 702. When FIFO break point (register) 832 is enabled, command processor 200 will not read past the CP_FIFO_BRK register. The CPU 100 can program this register 832 at the end of a frame. CPU 110 has to flush the write-buffer on the graphics and audio processor 114 and then read the FIFO write pointer 808. It then writes the value into the FIFO break register 832 and enables the break point.

If the size of the FIFO 210 is big enough to hold all the data sent in one frame, then the FIFO full condition shown in Figure 8B will never occur. However, this could mean allocating 2 to 4 Mbytes of main memory 112 for the FIFO buffer 210. Some application developers might not want to use that much memory for FIFO 210. In that case, the application should implement a flow control technique. Registers 826, 828 can be used to provide such flow control. Flow control is done in the example embodiment by having graphics and audio processor 114 generate an interrupt back to the CPU 110 when the number of cache lines in the main memory 110 surpasses FIFO_HICNT 826. The processor 110 will take the interrupt and

spin or do other non-graphical tasks, until the number of cache-lines in the FIFO is less than a FIFO_LOCNT 828. The reason for providing such a hysteresis effect is that interrupt overhead is high and one does not want to bounce in and out of the interrupt routine just by checking that the contents of the FIFO 210 has gone below the "high water mark". Interrupts can also be generated when the FIFO count goes below the LOCNT 828. This way, the application can perform other tasks and return when interrupted.

Example FIFO Buffer Allocation

In the preferred embodiment, the graphics API declares a static FXFifoObj structure internally. This structure is initialized when GXInit is called:

GXFifoObj* GXInit (void* base, u32 size);

The FIFO base pointer is aligned to 32b in the preferred embodiment. The application is responsible for allocating the memory for the FIFO. The size parameter for allocation is the size of the FIFO in bytes (the minimum FIFO size is 64KB, and size is a multiple of 32B). By default, GXInit sets up the FIFO for immediate mode graphics; that is: both the CPU 110 and graphics processor 114 are attached to the FIFO, the read and write pointers are initialized to the base pointer, and high and low water marks are enabled. GXInit returns a pointer to the initialized GXFifoObj to the application.

If the application wants to operate in multi-buffered mode, then additional FIFOs must be allocated. Any number of such additional FIFO buffers 210 can be allocated. The application allocates the memory for each additional FIFO and initializes a GXFifoObj as well. The following example functions can be used to initialize the GXFifoObj:

```
void GXInitFifoBase(
        GXFifoObj*   fifo,
        void*        base,
        u32          size);
void GXInitFifoPtrs(
        GXFifoObj*   fifo
        void*        read_ptr,
        void*        write_ptr );
void GXInitFifoLimits(
        GXFifoObj*   fifo,
        u32          hi_water_mark,
        u32          lo_water_mark );
```

Normally, the application only needs to initialize the FIFO read and write pointers to the base address of the FIFO. Once initialized, the system hardware will control the read and write pointers automatically.

Attaching and Saving FIFOs

Once a FIFO has been initialized, it can be attached to the CPU 110 or the graphics processor 114 or both. Only one FIFO may be attached to either the CPU 110 or graphics processor 114 at the same time. Once a FIFO is attached to the CPU 110, the CPU may issue GX commands to the FIFO. When a FIFO is attached to the graphics processor 114, it will be enabled to read graphics commands from the FIFO. The following example functions attach FIFOs:

```
void GXSetCPUFifo( GXFifoObj* fifo );
void GXSetGPFifo( GXFifoObj* fifo );
GXFifoObj* GXGetCPUFifo ( void ) ;
GXFifoObj* GXGetGPFifo ( void );
```

One may also inquire which FIFO objects are currently attached with these example functions:

```
GXFifoObj* GXGetCPUFifo ( void );
GXFifoObj* GXGetGPFifo ( void );
```

When in multi-buffer mode, and the CPU 110 is finished writing GX commands, the FIFO should be "saved" before switching to a new FIFO. The following example function "saves" the CPU FIFO:

```
void GXSaveCPUFifo ( FXFifoObj* fifo );
```

When a FIFO is saved, the CPU write-gather buffer 111 is flushed to make sure all graphics commands are written to main memory 112. In addition, the current FIFO read and write pointers are stored in the GXFifoObj structure.

Notice that there is no save function for the graphics processor 114. Once a graphics processor is attached, graphics commands will continue to be read until either:

- the FIFO is empty,
- a FIFO breakpoint is encountered, or
- the GP is pre-empted.

FIFO Status

The following example functions can be used to read the status of a FIFO and the GP:

```
void GXGetFifoStatus(
        GXFifoObj*   fifo,
        GXBool*      overhi,
        GXBool*      underlo,
        u32*         fifo_cnt,
        GXBool*      cpu_write,
        GXBool*      gp_read,
        GXBool*      fifowrap );
```

```
void GXGetGPStatus(
        GXBool*     overhi,
        GXBool*     underlow,
        GXBool*     readIdle,
        GXBool*     cmdIdle,
        GXBool*     brkpt );
```

GXGetFifoStatus gets the status of a specific FIFO. If the FIFO is currently attached to the CPU 110, the parameter cpu_write will be GX_TRUE. When the FIFO is currently attached to the graphics processor 114, the parameter gp_read will be GX_TRUE. When a FIFO is attached to either the CPU 110 or the graphics processor 114, the status will be read directly from the hardware's state. If the FIFO is not attached, the status will be read from the GXFifoObj. GXGetFifoStatus reports whether the specified FIFO has over flowed or has enough room to be written to. In general, the hardware cannot detect when a FIFO overflows, i.e., when the amount of data exceeds the size of the FIFO.

Although there is no general way to detect FIFO overflows, the hardware can detect when the CPU write pointer reaches the top of the FIFO. If this condition has occurred, the "fifowrap" argument will return GX_TRUE. The "fifowrap" argument can be used to detect FIFO overflows if the CPU's write pointer is always initialized to the base of the FIFO. "fifowrap" is set if the FIFO is currently attached to the CPU 110.

GXGetGPStatus can be used to get the status of the graphics processor 114 (regardless of the FIFO that attached to it). The minimum requirement to meet before attaching a new graphics processor FIFO is to wait for the graphics processor 114 to be idle (but additional constraints may also exist). The underlow and overhi

statuses indicate where the write pointer is, relative to the high and low water marks.

Example FIFO Flow Control

When a FIFO is attached to both the CPU and GP (immediate mode), care must be taken so that the CPU 110 stops writing commands when the FIFO is too full. A "high water mark" defines how full the FIFO can get before graphics commands will no longer be written to the FIFO. In the preferred embodiment, there may be up to 16KB of buffered graphics commands in the CPU, so it is recommended to set the high water mark to the (FIFO size – 16KB).

When the high water mark is encountered, the program will be suspended, but other interrupt-driven tasks such as audio will still be service. The programmer may also wish to specify which particular thread in a multi-threaded program should be suspended.

A "low water mark" defines how empty the FIFO must get after reaching a "high water mark" before the program (or thread) is allowed to continue. The low water mark is recommended to be set to (FIFO size / 2). The low water mark prevents frequent context switching of the program, since it does not need to poll some register or constantly receive overflow interrupts when the amount of new command data stays close to the high water mark.

When in multi-buffered mode, the high and low water marks are disabled. When a FIFO is attached to the CPU 110, and the CPU writes more commands than the FIFO will hold, the write pointer will be wrapped from the last address back to the base address. Previous graphics commands in the FIFO will be overwritten. It is possible to detect when the write pointer wraps over the top of the FIFO (which

indicates an overflow only if the FIFO write pointer was initialized to the base of the FIFO before commands were sent). See GXGetFifoStatus above.

In order to prevent FIFO (buffer) overflow in multi-buffered mode, a software-based checking scheme may be used. The program running on main processor 110 should keep its own counter of the buffer size, and before any group of commands is added to the buffer, the program may check and see if there is room. If room is available, the size of the group may be added to the buffer size. If room is not available, the buffer may be flushed and a new one allocated.

Using Display List Calls

To call a display list from a FIFO buffer 210 in the preferred embodiment, the application first allocates space in memory in which to store the display list. Once the memory area has been set up, the application can then call for example:

```
void GXBeginDisplayList (
        void        *list
        u32         size);
```

Where the "list" argument is the starting address for where the display list will be stored and the "size" argument indicates the number of bytes available in the allocated space for writing display list commands to allow the system to check for overflow.

Once "GXBeginDisplayList" has been called, further GX commands are written to the display list instead of to the normal command FIFO. The "GXEndDisplayList" command signals the end of the display list, and it returns the command steam to the FIFO to which it had been directed previously. The

"GXEndDisplayList" command also returns the actual size of the created display list as a multiple of 32 bytes in the example embodiment.

In the example embodiment. display lists cannot be nested. This means that once a GXBeginDisplayList has been issued, it is illegal to issue either another GXBeginDisplayLit or a GXCallDisplayList command until a GXEndDisplayList command comes along. However. in alternate embodiments it would be possible to provide display list nesting to any desired nesting level.

Example Graphics FIFO Functions

The following example functions provide management of the graphics FIFO:

GXSetFifoBase:

Argument:

u32	BasePtr;	//Set base address of fifo in main memory.
u32	Size;	//Size of the fifo in bytes. (a 32 bytes multiple).
GXBool	Set Defaults	//Setup default fifo state.

Sets the graphics fifo limits. This function is called at initialization time. The fifo address can not be changed unless the graphics pipe is flushed. If SetDefault flag is set, then the fifo is reset (i.e., read/write pointers at fifo base) and interrupts are disabled. By default, the high water mark is set to 2/3 of the size and the low water mark is set to 1/3 of the size.

GXSetFifoLimits:

Argument

u32	HiWaterMark;	//Hi-water mark for the fifo.
u32	LoWaterMark;	//Low water mark.
u32	RdBreakMark;	//Read pointer break point.

This function sets the fifo limits. When the read pointer goes below low water mark or when write pointer goes above high water mark, the graphics hardware will interrupt the CPU. The RdBreakMark is used for setting read pointer break point.

GXSetInterrupts:

Argument

```
GXBool  Underflow;   //Enable/Disable low water mark interrupt.
GXBool  Overflow;    //Enable/Disable high water mark interrupt.
GXBool  BreakPoint;  //Enable/Disable fifo read break point.
```

Enables or disables fifo related interrupts. The BreakPoint is a feature than can be used to halt fifo reads by the CP while a previous frame is still being copied.

GXClearInterrupts:

Argument:

```
GXBool  Underflow;   //Clear low water mark interrupt
GXBool  Overflow;    //Clear high water mark interrupt.
GXBool  BreakPoint   //Clear fifo read break point.
```

Clears a pending interrupt.

GXSetFifoPtrs:

Argument:

```
u32     WritePtr;    //Sets write pointer for fifo.
u32     ReadPtr;     //Sets read pointer.
```

Sets fifo read and write pointers. These pointers are maintained by the hardware. This function will override the hardware values (e.g., for display list compilation).

GXGetFifoStatus:

Argument:

```
GXBool  *UnderFlow;   //Fifo count is below low water mark.
GXBool  *OverFlow;    //Fifo count is above high water mark.
GXBool  *BreakPoint;  //Fifo read pointer is at break point.
u32     *FifoCount;   //Number of cachelines (32 bytes) in Fifo.
```

Returns fifo status and count.

Example Display List Functions

A display list is an array of pre-compiled commands and data for the graphics pipe. The following example commands are inserted into a FIFO buffer 210 to manipulate display lists.

GXBeginDisplayList:

Argument:

```
void*   BasePtr;   //Address of a buffer in for storing display
                   list data.
u32     nBytes;    //Size of the buffer.
```

This function creates and starts a display list. The API is put in display list mode. All API functions, except any of the display list functions, following this call until EndDisplayList, send their data and commands to the display list buffer instead of graphics pipe. A display list can not be nested in this example, i.e., no display list functions can be called between a BeginDisplayList and EndDisplayList. The memory for the display list is allocated by the application.

GXEndDisplayList:

Argument:

None.

Return:

u32 nBytes //Number of bytes used for the display list.

This function ends currently opened display object and puts the system back in immediate mode.

GXCallDisplayList:

Argument:

```
void*   BasePtr;   //Address of a buffer in for storing display list data.
u32     nBytes;    //Size of the buffer
```

This function executes the display list.

Example Register Formats:

The following table shows example registers in the command processor 200 that are addressable by CPU 110:

Register Name	Bit Fields: Description
CP_STATUS Register 834	0: FIFO overflow (fifo_count > FIFO_HICNT)
	1: FIFO underflow (fifo_count < FIFO_LOCNT)
	2: FIFO read unit idle
	3: CP idle
	4: FIFO reach break point (cleared by disable FIFO break point)

Register Name	Bit Fields:	Description
CP_ENABLE Register 836	0:	Enable FIFO reads, reset value is "0" disable
	1:	FIFO break point enable bit, reset value is "0" disable
	2:	FIFO overflow interrupt enable, reset value is "0" disable
	3:	FIFO underflow interrupt enable, reset value is "0" disable
	4:	FIFO write pointer increment enable, reset value is "1" enable
	5:	FIFO break point interrupt enable, reset value is "0" disable
CP_CLEAR Register 838	0:	clear FIFO overflow interrupt
	1:	clear FIFO underflow interrupt
CP_STM_LOW Register 840	7:0	bits 7:0 of the Streaming Buffer low water mark in 32 bytes increment, default (reset) value is "0x0000"
CP_FIFO_BASEL 822	15:5	bits 15:5 of the FIFO base address in memory
CP_FIFO_BASE 822	9:0	bits 25:16 of the FIFO base address in memory
CP_FIFO_TOPL 824	15:5	bits 15:5 of the FIFO top address in memory
CP_FIFO_TOPH 824	9:0	bits 25:16 of the FIFO top address in memory
CP_FIFO_HICNTL 826	15:5	bits 15:5 of the FIFO high water count
CP_FIFO_HICNTH 826	9:0	bits 25:16 of the FIFO high water count
CP_FIFO_LOCNTL 828	15:5	bits 15:5 of the FIFO low water count
CP_FIFO_LOCNTH 828	9:0	bits 25:16 of the FIFO low water count
CP_FIFO_COUNTL 830	15:5	bits 15:5 of the FIFO_COUNT (entries currently in FIFO)
CP_FIFO_COUNTH 830	9:0	bits 25:16 of the FIFO_COUNT (entries currently in FIFO)
CP_FIFO_WPTRL 808	15:5	bits 15:5 of the FIFO write pointer
CP_FIFO_WPTRH 808	9:0	bits 25:15 of the FIFO write pointer
CP_FIFO_RPTRL 804	15:5	bits 15:5 of the FIFO read pointer
CP_FIFO_RPTRH 804	9:0	bits 25:15 of the FIFO read pointer
CP_FIFO_BRKL 832	15:5	bits 15:5 of the FIFO read address break point
CP_FIFO_BRKH 832	9:0	bits 9:0 if the FIFO read address break point

Other Example Compatible Implementations

Certain of the above-described system components 50 could be implemented as other than the home video game console configuration described above. For example, one could run graphics application or other software written for system 50

on a platform with a different configuration that emulates system 50 or is otherwise compatible with it. If the other platform can successfully emulate, simulate and/or provide some or all of the hardware and software resources of system 50, then the other platform will be able to successfully execute the software.

As one example, an emulator may provide a hardware and/or software configuration (platform) that is different from the hardware and/or software configuration (platform) of system 50. The emulator system might include software and/or hardware components that emulate or simulate some or all of hardware and/or software components of the system for which the application software was written. For example, the emulator system could comprise a general purpose digital computer such as a personal computer, which executes a software emulator program that simulates the hardware and/or firmware of system 50.

Some general purpose digital computers (e.g., IBM or MacIntosh personal computers and compatibles) are now equipped with 3D graphics cards that provide 3D graphics pipelines compliant with OpenGL, DirectX or other standard 3D graphics command APIs. They may also be equipped with stereophonic sound cards that provide high quality stereophonic sound based on a standard set of sound commands. Such multimedia-hardware-equipped personal computers running emulator software may have sufficient performance to approximate the graphics and sound performance of system 50. Emulator software controls the hardware resources on the personal computer platform to simulate the processing, 3D graphics, sound, peripheral and other capabilities of the home video game console platform for which the game programmer wrote the game software.

Figure 12A illustrates an example overall emulation process using a host platform 1201, an emulator component 1303, and a game software executable

binary image provided on a storage medium 62. Host 1201 may be a general or special purpose digital computing device such as, for example, a personal computer, a video game console, or any other platform with sufficient computing power. Emulator 1303 may be software and/or hardware that runs on host platform 1201, and provides a real-time conversion of commands, data and other information from storage medium 62 into a form that can be processed by host 1201. For example, emulator 1303 fetches "source" binary-image program instructions intended for execution by system 50 from storage medium 62 and converts these program instructions to a target format that can be executed or otherwise processed by host 1201.

As one example, in the case where the software is written for execution on a platform using an IBM PowerPC or other specific processor and the host 1201 is a personal computer using a different (e.g., Intel) processor, emulator 1303 fetches one or a sequence of binary-image program instructions from storage medium 1305 and converts these program instructions to one or more equivalent Intel binary-image program instructions. The emulator 1303 also fetches and/or generates graphics commands and audio commands intended for processing by the graphics and audio processor 114, and converts these commands into a format or formats that can be processed by hardware and/or software graphics and audio processing resources available on host 1201. As one example, emulator 1303 may convert these commands into commands that can be processed by specific graphics and/or or sound hardware of the host 1201 (e.g., using standard DirectX, OpenGL and/or sound APIs).

Certain emulators of system 50 might simply "stub" (i.e., ignore) some or all of the buffering and flow control techniques described above since they might have

much more memory resources than the example hardware implementation described above. Such emulators will typically respond to requests for buffer allocation by allocating memory resources, but might provide different flow control processing. Status and flow control requests as described above could be emulated by maintaining an emulated state of the hardware, and using that state to respond to the status requests.

An emulator 1303 used to provide some or all of the features of the video game system described above may also be provided with a graphic user interface (GUI) that simplifies or automates the selection of various options and screen modes for games run using the emulator. In one example, such an emulator 1303 may further include enhanced functionality as compared with the host platform for which the software was originally intended.

Figure 12B illustrates an emulation host system 1201 suitable for use with emulator 1303. System 1201 includes a processing unit 1203 and a system memory 1205. A system bus 1207 couples various system components including system memory 1205 to processing unit 1203. System bus 1207 may be any of several types of bus structures including a memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of bus architectures. System memory 1207 includes read only memory (ROM) 1252 and random access memory (RAM) 1254. A basic input/output system (BIOS) 1256, containing the basic routines that help to transfer information between elements within personal computer system 1201, such as during start-up, is stored in the ROM 1252. System 1201 further includes various drives and associated computer-readable media. A hard disk drive 1209 reads from and writes to a (typically fixed) magnetic hard disk 1211. An additional (possible optional) magnetic disk drive 1213 reads from and writes to a

removable "floppy" or other magnetic disk 1215. An optical disk drive 1217 reads from and, in some configurations, writes to a removable optical disk 1219 such as a CD ROM or other optical media. Hard disk drive 1209 and optical disk drive 1217 are connected to system bus 1207 by a hard disk drive interface 1221 and an optical drive interface 1225, respectively. The drives and their associated computer-readable media provide nonvolatile storage of computer-readable instructions, data structures, program modules, game programs and other data for personal computer system 1201. In other configurations, other types of computer-readable media that can store data that is accessible by a computer (e.g., magnetic cassettes, flash memory cards, digital video disks, Bernoulli cartridges, random access memories (RAMs), read only memories (ROMs) and the like) may also be used.

A number of program modules including emulator 1303 may be stored on the hard disk 1211, removable magnetic disk 1215, optical disk 1219 and/or the ROM 1252 and/or the RAM 1254 of system memory 1205. Such program modules may include an operating system providing graphics and sound APIs, one or more application programs, other program modules, program data and game data. A user may enter commands and information into personal computer system 1201 through input devices such as a keyboard 1227, pointing device 1229, microphones, joysticks, game controllers, satellite dishes, scanners, or the like. These and other input devices can be connected to processing unit 1203 through a serial port interface 1231 that is coupled to system bus 1207, but may be connected by other interfaces, such as a parallel port, game port Fire wire bus or a universal serial bus (USB). A monitor 1233 or other type of display device is also connected to system bus 1207 via an interface, such as a video adapter 1235.

System 1201 may also include a modem 1154 or other network interface means for establishing communications over a network 1152 such as the Internet. Modem 1154, which may be internal or external, is connected to system bus 123 via serial port interface 1231. A network interface 1156 may also be provided for allowing system 1201 to communicate with a remote computing device 1150 (e.g., another system 1201) via a local area network 1158 (or such communication may be via wide area network 1152 or other communications path such as dial-up or other communications means). System 1201 will typically include other peripheral output devices, such as printers and other standard peripheral devices.

In one example, video adapter 1235 may include a 3D graphics pipeline chip set providing fast 3D graphics rendering in response to 3D graphics commands issued based on a standard 3D graphics application programmer interface such as Microsoft's DirectX 7.0 or other version. A set of stereo loudspeakers 1237 is also connected to system bus 1207 via a sound generating interface such as a conventional "sound card" providing hardware and embedded software support for generating high quality stereophonic sound based on sound commands provided by bus 1207. These hardware capabilities allow system 1201 to provide sufficient graphics and sound speed performance to play software stored in storage medium 62.

All documents referenced above are hereby incorporated by reference.

While the invention has been described in connection with what is presently considered to be the most practical and preferred embodiment, it is to be understood that the invention is not to be limited to the disclosed embodiment, but on the contrary, is intended to cover various modifications and equivalent arrangements included within the scope of the appended claims.